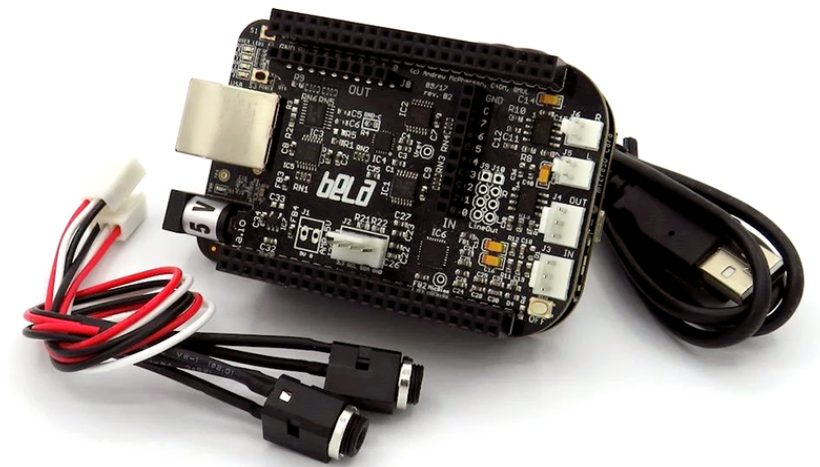
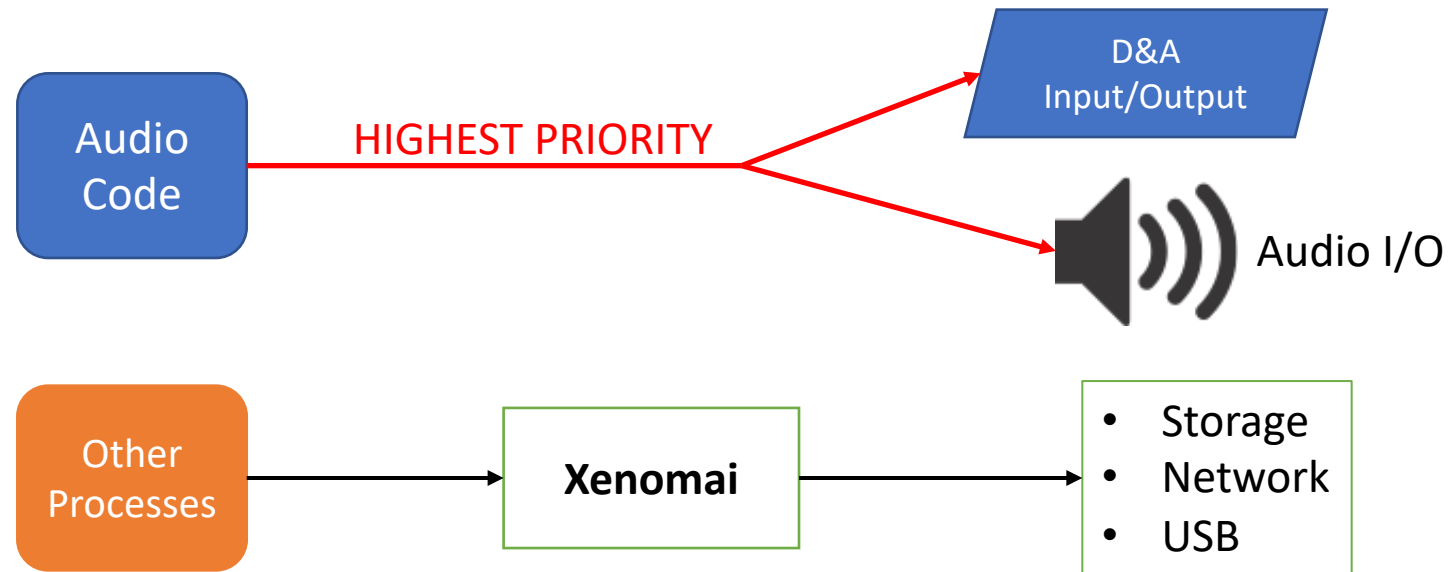




# The Bela board and Framework

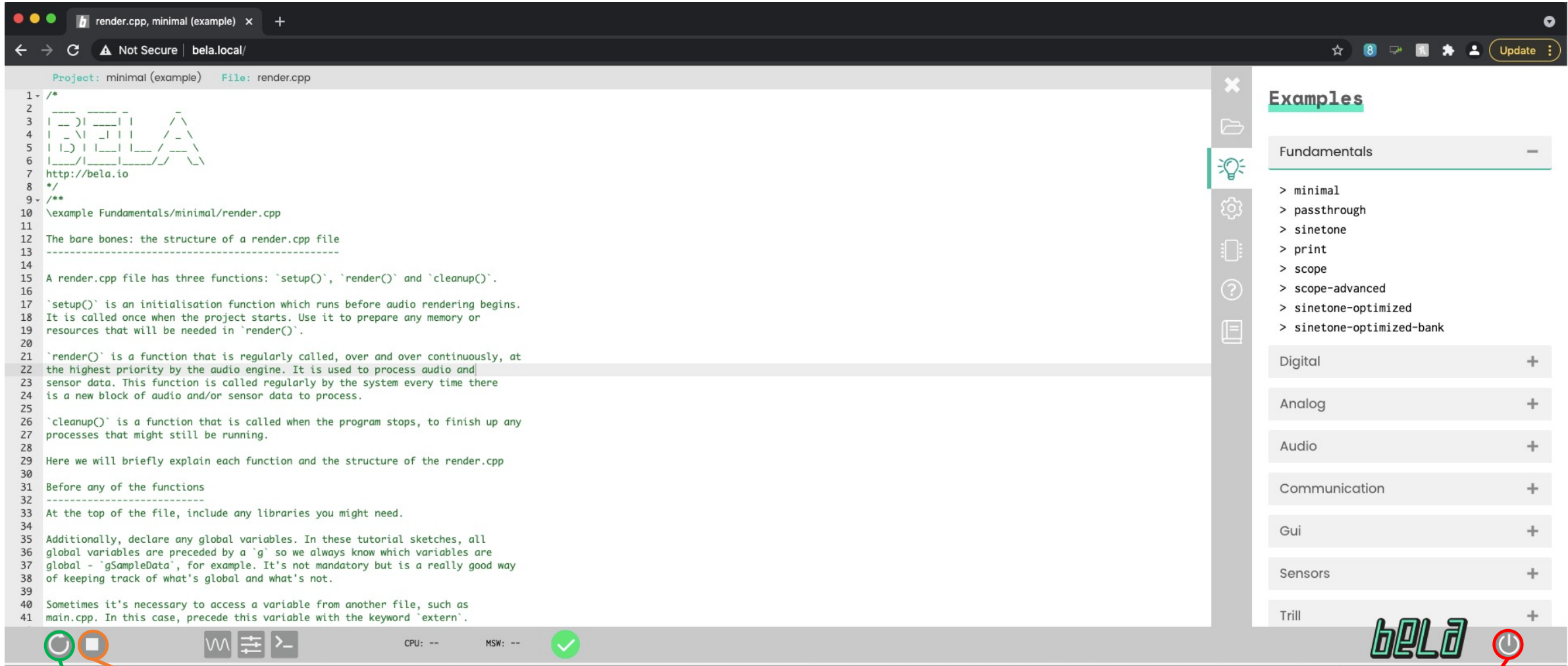
- Bela is an embedded computer design for interactive real-time audio
- Offers both the power of a board computer and the connectivity of a microcontroller
- Bela uses the Linux Xenomai OS to run real-time audio code at the highest priority





- Its architecture is optimised for real time audio and can provide a latency as low as 100µs
- Bela is designed for audio and therefore offers embedded ADCs/DACs and embedded Audio I/Os
- Bela offers its own programming framework and provides a series of libraries for the design of real-time audio interaction
- Bela can be programmed in three different languages:
  1. C++
  2. Pure Data -> great for prototyping and with a huge availability of open-source libraries
  3. Super Collider -> audio dedicated DSP language. Offers real-time coding
- Bela can be used to design programmed hardware applications such as standalone synth, FX pedals, modules for modular synths, FX units, etc...
- Provides embedded MIDI support via regular USB

The IDE is browser based and can be accessed at *bela.local/* (Chrome is recommended by manufacturer)



The screenshot shows the Bela IDE web interface in a browser. The address bar shows `bela.local/`. The main editor displays a C++ file named `render.cpp` with a minimal example. The sidebar on the right lists various examples under the heading "Examples". The bottom status bar contains several icons: a green circle with a white play button (Run), a red square with a white stop button (Stop), a green checkmark (Success), and a power button icon (Shutdown). The "bela" logo is also visible in the bottom right corner of the IDE interface.

Project: minimal (example) File: render.cpp

```
1- /*
2- 3- 4- 5- 6- 7- http://bela.io
8- 9- 10- \example Fundamentals/minimal/render.cpp
11- 12- The bare bones: the structure of a render.cpp file
13- 14- 15- A render.cpp file has three functions: 'setup()', 'render()' and 'cleanup()'.
16- 17- 'setup()' is an initialisation function which runs before audio rendering begins.
18- 19- It is called once when the project starts. Use it to prepare any memory or
20- 21- resources that will be needed in 'render()'.
22- 23- 'render()' is a function that is regularly called, over and over continuously, at
24- 25- the highest priority by the audio engine. It is used to process audio and
26- 27- sensor data. This function is called regularly by the system every time there
28- 29- is a new block of audio and/or sensor data to process.
30- 31- 'cleanup()' is a function that is called when the program stops, to finish up any
32- 33- processes that might still be running.
34- 35- Here we will briefly explain each function and the structure of the render.cpp
36- 37- Before any of the functions
38- 39- -----
40- 41- At the top of the file, include any libraries you might need.
42- 43- Additionally, declare any global variables. In these tutorial sketches, all
44- 45- global variables are preceded by a 'g' so we always know which variables are
46- 47- global - 'gSampleData', for example. It's not mandatory but is a really good way
48- 49- of keeping track of what's global and what's not.
50- 51- Sometimes it's necessary to access a variable from another file, such as
52- 53- main.cpp. In this case, precede this variable with the keyword 'extern'.
```

Examples

Fundamentals

- > minimal
- > passthrough
- > sinetone
- > print
- > scope
- > scope-advanced
- > sinetone-optimized
- > sinetone-optimized-bank

Digital +

Analog +

Audio +

Communication +

Gui +

Sensors +

Trill +

Run program

Stop program

Shutdown Bela board



When working in C++, every Bela program has three main functions:

1) **bool *setup*(BelaContext \*context, void \*userData)**

- Run once only at the start; before audio start
- The structure *context* holds all the info on channels sample rate, block sizes etc...
- Initialises the program: if successful return *true* otherwise false to stop the program

2) **void *render*(BelaContext \*context, void \*userData)**

- This is the **Audio callback** function
- Called for every new block of samples → Bela process sound in blocks of sample not sample by sample
- Where the core code goes!

3) **void *cleanup*(BelaContext \*context, void \*userData)**

- Is run once at the end and cleans up all the resources allocated in *setup()*

A significant number of operations are carried out by accessing *BelaContext*

- *BelaContext* is a data structure embedded in the framework which holds most of the information needed for the audio processing:
  - Block sizes -> Bela works on 16 samples block size by default: **int audioFrames**
  - Sample rate: **float audioSampleRate**
  - Number of input channels -> Bela has a stereo input **int audioInChannels**
  - Number of output channels -> Bela has a stereo output **int audioOutChannels**
  - The actual sample data:

**float\* audioIn**

L0	L1	L2	L3	L4	L5	L6	L7
R0	R1	R2	R3	R4	R5	R6	R7

Audio input buffer for all channels (stereo)

**float\* audioOut**

L0	L1	L2	L3	L4	L5	L6	L7
R0	R1	R2	R3	R4	R5	R6	R7

Audio output buffer for all channels (stereo)



Let's take a look back at the audio callback function:

```
void render(BelaContext *context, void *userData)
```

- *render()* gets a pointer to the data structure so all the information contained in *BelaContext* are passed to the callback.

```
void render(BelaContext *context, void *userData)
{
    . . .
    context -> audioSampleRate;
    context -> audioFrames;
    . . .
}
```

- As *render()* gets a pointer all the information stored inside the structure can be accessed through the arrow operator.



Access the output buffer, a convenience function:

```
void audioWrite(BelaContext *context, int frame, int channel, float value)
```

Reference to BelaContext

Which sample (frame)  
within the buffer to write

Which channel (L/R) to  
write

What value to write to  
write





Access the output buffer, a convenience function:

```
void audioWrite(BelaContext *context, int frame, int channel, float value)
```

Reference to BelaContext

Which sample (frame)  
within the buffer to write

Which channel (L/R) to  
write

What value to write to  
write

There is another convenience function to access the input buffer which is known as *audioRead* but we will give a look at that later...



## Let's take a look at some code (*sine\_gen\_5s.cpp*)

```
#include <Bela.h>
#include <cmath>
#include <vector>
#include <libraries/AudioFile/AudioFile.h>

//This code does not generate audio in real-time. It produces a 5 second sine wave which is then play in a loop
//the 5 seconds sine wave is also recorded on a .wav file

float gFrequency = 440.0;      // Frequency of the sine wave in Hz
float gAmplitude = 0.6;       // Amplitude of the sine wave (1.0 is maximum)
float gDuration = 5.0;        // Length of file to generate, in seconds

std::string gFilename = "output.wav"; //Names the file on which the sine will be saved
std::vector<float> gSampleData; //Creates a vector to store the sine wave data
int gNumSamples = 0;
int gReadPointer = 0;

// This function calculates a sine wave, storing it in a buffer of length numSamples.
void calculate_sine(float *buffer, int numSamples, float sampleRate)
{
    // Generate the sine wave sample-by-sample for the whole duration
    for (int n = 0; n < numSamples; n++) {
        // Calculate one sample of the sine wave
        float out = gAmplitude * sin(2.0 * M_PI * n * gFrequency / sampleRate);
        // Store the sample in the buffer
        buffer[n] = out;
    }
}

bool setup(BelaContext *context, void *userData)
{
    float sampleRate = context->audioSampleRate;      //Sample Rate is taken from the BelaContext structure
    gNumSamples = gDuration * sampleRate;             //Duration*sampleRate = number of total samples

    gSampleData.resize(gNumSamples);

    calculate_sine(gSampleData.data(), gNumSamples, sampleRate);

    AudioFileUtilities::write(gFilename, gSampleData.data(), 1, gNumSamples, sampleRate); //Writes the buffer

    return true;
}

void render(BelaContext *context, void *userData) //Callback funtion
{
    for(unsigned int n = 0; n < context->audioFrames; n++) {
        // Increment read pointer and reset to 0 when end of file is reached
        if(++gReadPointer >= gNumSamples)
            gReadPointer = 0;

        for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
            float out = gSampleData[gReadPointer];
            audioWrite(context, n, channel, out); // Write the sample to every audio output channel (1=left, 2=right)
        }
    }
}

void cleanup(BelaContext *context, void *userData)
{
}
```



Let's take a look at some code (*sine\_gen\_5s.cpp*)

- The sine generator we just looked at does not produce audio in real time. It calculates 5 seconds of sine wave at once and plays it back
- How do we get real-time?

Let's take a look at some code (*sine\_gen\_5s.cpp*)

- The sine generator we just looked at does not produce audio in real time. It calculates 5 seconds of sine wave at once and plays it back
- How do we get real-time?

**Possible solution:** run a portion of code anytime we need a new sample for the sine wave...Will this guarantee real-time?

- For a sampling rate of 44100Hz and a frequency of 440Hz we would need a new sample every 22.7μs...



Let's take a look at some code (*sine\_gen\_5s.cpp*)

- The sine generator we just looked at does not produce audio in real time. It calculates 5 seconds of sine wave at once and plays it back
- How do we get real-time?

**Possible solution:** run a portion of code anytime we need a new sample for the sine wave..Will this guarantee real-time?

- For a sampling rate of 44100Hz and a frequency of 440Hz we would need a new sample every 22.7μs...

Let's go to the IDE and type *ps* in the console...



Let's take a look at some code (*sine\_gen\_5s.cpp*)

- The sine generator we just looked at does not produce audio in real time. It calculates 5 seconds of sine wave at once and plays it back
- How do we get real-time?

Possible solution: run a portion of code anytime we need a new sample for the sine wave..Will this guarantee real-time?

- For a sampling rate of 44100Hz and a frequency of 440Hz we would need a new sample every 22.7μs...

Let's go to the IDE and type *ps* in the console...

**DUE TO THE AMOUNT OF PROCESSES HAPPENING ON THE BOARD, WE CANNOT GUARANTEE US THAT THE AUDIO CALCULATION WILL HAPPEN ON TIME!!**



## THAT IS WHY BELA WORKS ON BLOCK-BASED PROCESSING!!

Better solution: process the sound in blocks of several samples each!

- The typical **block size** in Bela is between 2 and 32 samples with a **default of 16**



## THAT IS WHY BELA WORKS ON BLOCK-BASED PROCESSING!!

Better solution: process the sound in blocks of several samples each!

- The typical **block size** in Bela is between 2 and 32 samples with a **default of 16**

Why is block based processing better for real-time?

...



## THAT IS WHY BELA WORKS ON BLOCK-BASED PROCESSING!!

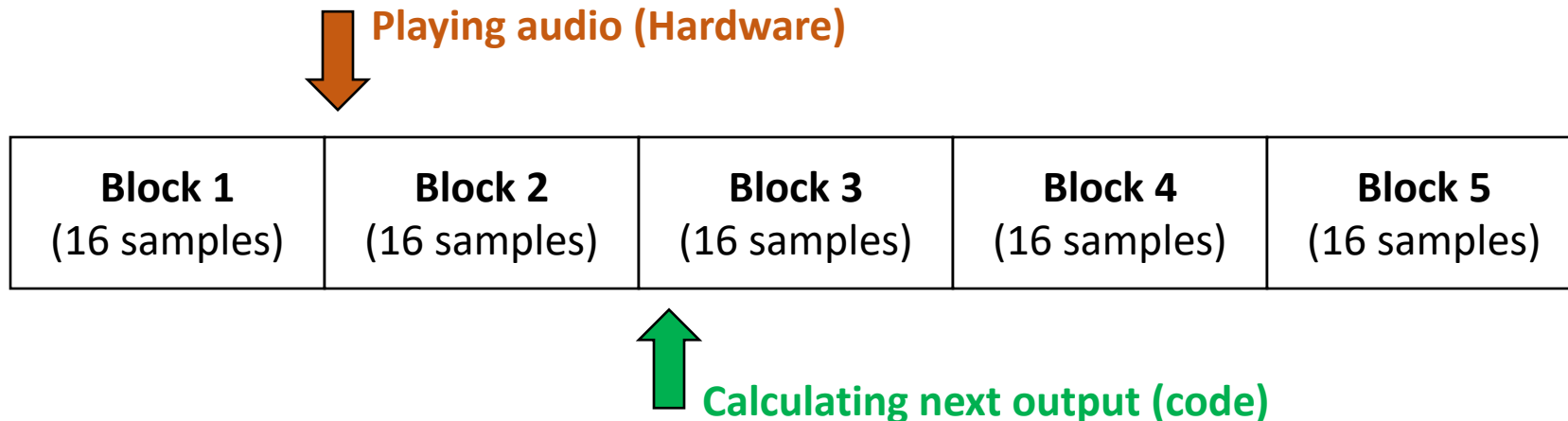
Better solution: process the sound in blocks of several samples each!

- The typical **block size** in Bela is between 2 and 32 samples with a **default of 16**

Why is block based processing better for real-time?

...

While the hardware is playing one block of samples...the code will calculate the next!!



## Let's take a look at some code (*sine\_gen\_rt\_n.cpp*)

```
#include <Bela.h>
#include <cmath>

float gFrequency = 440.0;    // Frequency of the sine wave in Hz
float gAmplitude = 0.6;     // Amplitude of the sine wave (1.0 is maximum)
unsigned int gTotalSample = 0;

bool setup(BelaContext *context, void *userData)
{
    return true;
}

void render(BelaContext *context, void *userData)
{
    // This for() loop goes through all the samples in the block
    for (unsigned int n = 0; n < context->audioFrames; n++) {

        float out = gAmplitude * sin(2.0 * M_PI * n * gFrequency / context->audioSampleRate);

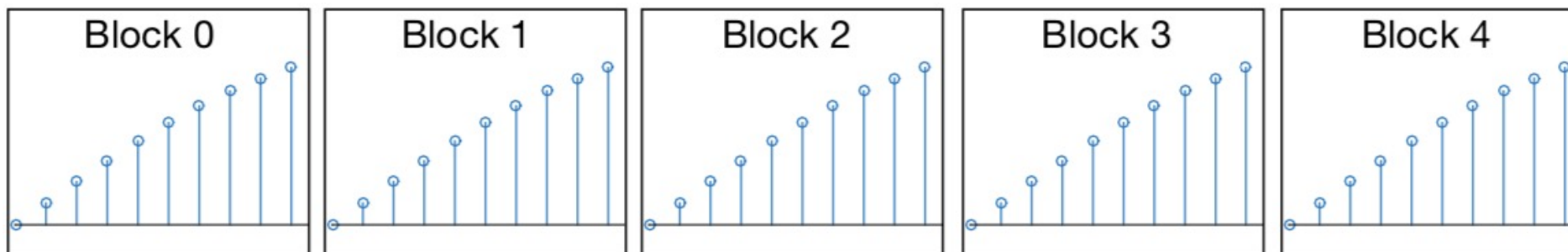
        for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
            audioWrite(context, n, channel, out);
        }
    }
    //gFrequency *= 1.00001; //Increase the frequency gradually
}

void cleanup(BelaContext *context, void *userData)
{
    // Nothing to do here
}
```

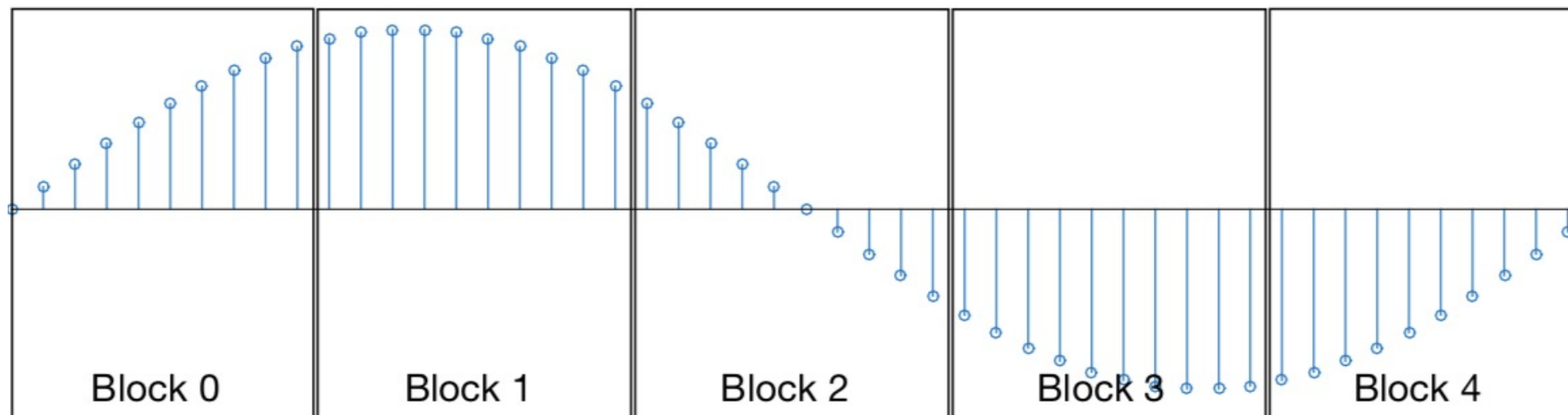


Let's take a look at some code (*sine\_gen\_rt\_n.cpp*)

- What we are getting:



- What we want:



## We need to **preserve state** between calls to render()

- When render() runs a second time, it should remember when it left off the first time
- But local variables in the function all disappear when the function returns!
- Solution: use **global variables** to save the state

## Let's take a look at some code (*sine\_gen\_rt\_totsam.cpp*)

```
#include <Bela.h>
#include <cmath>

float gFrequency = 440.0; // Frequency of the sine wave in Hz
float gAmplitude = 0.6; // Amplitude of the sine wave (1.0 is maximum)
unsigned int gTotalSample = 0;

bool setup(BelaContext *context, void *userData)
{
    return true;
}

void render(BelaContext *context, void *userData)
{
    // This for() loop goes through all the samples in the block
    for (unsigned int n = 0; n < context->audioFrames; n++) {

        gTotalSample++;

        float out = gAmplitude * sin(2.0 * M_PI * gTotalSample * gFrequency / context->audioSampleRate);
        //NOTE: tots sample instead of n

        for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
            audioWrite(context, n, channel, out);
        }
    }
    //gFrequency *= 1.00001; //Increase the frequency gradually
}

void cleanup(BelaContext *context, void *userData)
{
    // Nothing to do here
}
```



## What about a variable frequency?

Let's take a look at some code (*sine\_gen\_rt\_totsam.cpp*) --> UNCOMMENT LINE 25!

```
#include <Bela.h>
#include <cmath>

float gFrequency = 440.0;           // Frequency of the sine wave in Hz
float gAmplitude = 0.6;             // Amplitude of the sine wave (1.0 is maximum)
unsigned int gTotalSample = 0;

bool setup(BelaContext *context, void *userData)
{
    return true;
}

void render(BelaContext *context, void *userData)
{
    // This for() loop goes through all the samples in the block
    for (unsigned int n = 0; n < context->audioFrames; n++) {

        gTotalSample++;

        float out = gAmplitude * sin(2.0 * M_PI * gTotalSample * gFrequency / context->audioSampleRate);
        //NOTE: tots sample instead of n

        for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
            audioWrite(context, n, channel, out);
        }

        gFrequency *= 1.00001; //Increase the frequency gradually
    }
}

void cleanup(BelaContext *context, void *userData)
{
    // Nothing to do here
}
```

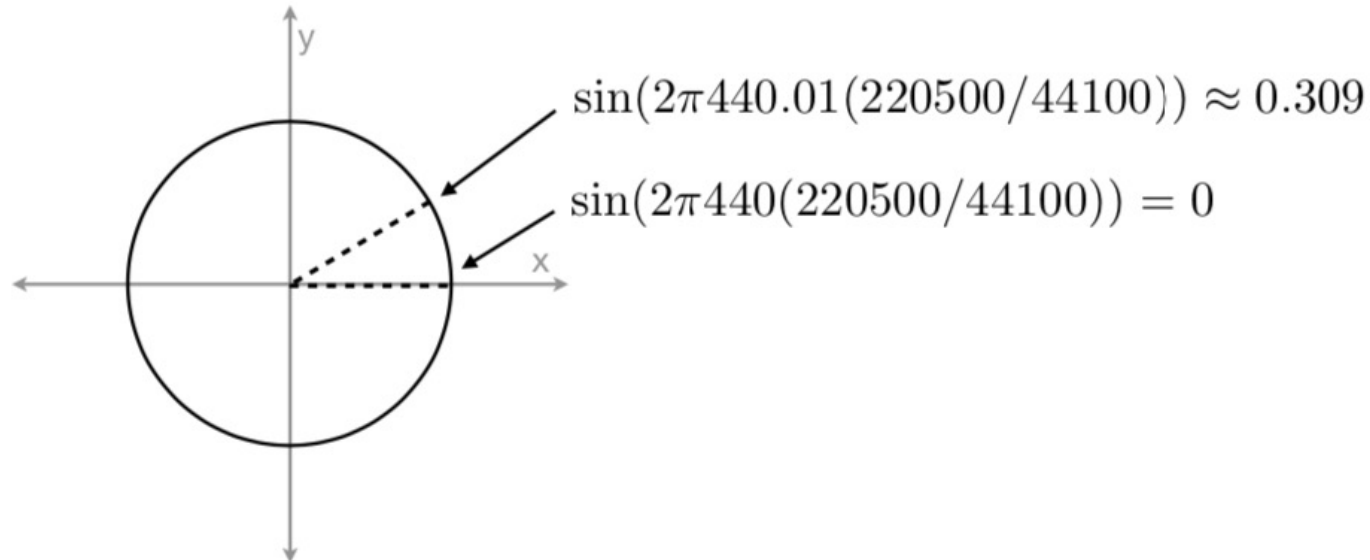
## What about a variable frequency?

Let's take a look at some code (*sine\_gen\_rt\_totsam.cpp*) --> UNCOMMENT LINE 25!

...

...

- Frequency is the derivative of phase (argument of the sine function)
- ↓
- We recalculate the phase every time based on how many samples have gone by
- ↓
- If we change the frequency, the phase can jump, which causes the problem



**We need to remember  
the phase!!**

As per the unit circle a sine wave oscillates between 0 and  $2\pi$ ...currently the phase is exceeding  $2\pi$  and needs therefore to be wrapped at the end of every period...

Let's take a look at some code (*sine\_gen\_rt\_phase.cpp*)

```
#include <Bela.h>
#include <cmath>

float gFrequency = 440.0;           // Frequency of the sine wave in Hz
float gAmplitude = 0.6;             // Amplitude of the sine wave (1.0 is maximum)
float gPhase = 0;

bool setup(BelaContext *context, void *userData)
{
    return true;
}

void render(BelaContext *context, void *userData)
{
    // This for() loop goes through all the samples in the block
    for (unsigned int n = 0; n < context->audioFrames; n++) {

        //PHASE WRAPPING!
        gPhase += (2.0 * M_PI * gFrequency / context->audioSampleRate);
        if(gPhase >= 2*M_PI)
            gPhase -= 2*M_PI;

        float out = gAmplitude * sin(gPhase);

        for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
            audioWrite(context, n, channel, out);
        }

        gFrequency *= 1.00001; //Increase the frequency gradually
    }

    // cleanup() runs once at the end of the program
    void cleanup(BelaContext *context, void *userData)
    {
        // Nothing to do here
    }
}
```

Another feature of Bela is multiple ways that can be employed to control parameters:

### 1. Virtual GUI

In Bela, we can create a GUI through code that will appear on a separate tab of the browser and give us control over the desired parameters.

Despite is very basic this can be very useful when in lack of MIDI controllers or analogue potentiometers

### 1. Analogue controls

Like most embedded prototyping boards, Bela offers a series of both analogue and digital I/O. A 5V volts signal can be generated from the board itself and fed to Bela through potentiometer.

The variation of voltage will be read from Bela and generate a 0-1 scalar to control chosen parameters

### 1. MIDI

Bela has built in MIDI and can receives MIDI messages through the main USB port.

A MIDI library is used in the code to determine the behaviour of the program.





## Let's have a look at the browser GUI...(*sine\_gen\_rt1\_gui.cpp*)

```
#include <Bela.h>
#include <cmath>

#include <libraries/Gui/Gui.h>
#include <libraries/GuiController/GuiController.h>

Gui gui;
GuiController controller;

float gPhase = 0;

bool setup(BelaContext *context, void *userData)
{
    //Setup GUI
    gui.setup(context->projectName);
    controller.setup(&gui, "Sine");
    controller.addSlider("Hz", 440, 150, 800, 0);
    controller.addSlider("Vol", 0.5, 0, 1, 0);

    return true;
}

void render(BelaContext *context, void *userData)
{
    float freq = controller.getSliderValue(0); //Assign GUI vaue to variable
    float amp = controller.getSliderValue(1); //Assign GUI vaue to variable

    for (unsigned int n = 0; n < context->audioFrames; n++) {

        gPhase += (2.0 * M_PI * freq / context->audioSampleRate); //The argument of the sine function is stored in a variable
        if(gPhase >= 2*M_PI)
            gPhase -= 2*M_PI;

        float out = amp * sin(gPhase); //Sine wave is calculated using the variable created in line 31

        for(unsigned int channel = 0; channel < context->audioOutChannels; channel++) {
            audioWrite(context, n, channel, out);
        }
    }
}

void cleanup(BelaContext *context, void *userData)
{
    // Nothing to do here
}
```



Let's set up MIDI to control the sine generator...(*sine\_gen\_rt1\_MIDI.cpp*)

We will need:

MIDI to Hz:

$$f = 440 * 2^{\frac{\text{note number} - 69}{12}}$$

MIDI velocity to amplitude in dB:

$$A_{dB} = 20 \log_{10}(A_{linear})$$

$$A_{linear} = 10^{\frac{A_{dB}}{20}}$$



**THANK YOU!**

Eugenio Donati